

MAC-DBT Revisited

Roie Zivan, Uri Shapen, Moshe Zazone, and Amnon Meisels
Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel
{zivanr,shapenko,moshezaz,am}@cs.bgu.ac.il

Abstract

Dynamic Backtracking (*DBT*) is a well known algorithm for solving Constraint Satisfaction Problems. In *DBT*, variables are allowed to keep their assignment during backjump, if they are compatible with the set of eliminating explanations. A previous study has shown that when *DBT* is combined with variable ordering heuristics, it performs poorly compared to standard Conflict-directed Backjumping (*CBJ*) [Bak94]. In later studies, *DBT* was enhanced with constraint propagation methods. The *MAC-DBT* algorithm was reported by [JDB00] to be the best performing version, improving on both standard *DBT* and on *FC-DBT* by a large factor.

The present study evaluates the *DBT* algorithm from a number of aspects. First we show that the advantage of *MAC-DBT* over *FC-DBT* holds only for a static ordering. When dynamic ordering heuristics are used, *FC-DBT* outperforms *MAC-DBT*. Second, we show theoretically that a combined version of *DBT* that uses both *FC* and *MAC* performs equal or less computation at each step than *MAC-DBT*. An empirical result which presents the advantage of the combined version on *MAC-DBT* is also presented. Third, following the study of [Bak94], we present a version of *MAC-DBT* and *FC-DBT* which does not preserve assignments which were jumped over. It uses the *Nogood* mechanism of *DBT* only to determine which values should be restored to the domains of variables. These versions of *MAC-DBT* and *FC-DBT* outperform all previous versions.

1 Introduction

Conflict Based Backjumping (*CBJ*) is a technique which is known to improve the search of Constraint Satisfaction Problems (*CSPs*) by a large factor [Dec03, KvB97, CvB01]. Its efficiency increases when it is combined with forward checking [Pro93]. The down side of *CBJ* is that when such a backtrack (back-jump) is performed, assignments of variables which were assigned later than the culprit assignment are discarded.

Dynamic Backtracking [Gin93] improves on standard *CBJ* by preserving assignments of non conflicting variables during back-jumps. In the original form of *DBT*, the culprit variable which replaces its assignment is moved to be the last among the assigned variables. In other words, the new assignment of the culprit variable must be consistent with all former assignments [Gin93].

Although *DBT* saves unnecessary assignment attempts and was proposed as an improvement to *CBJ*, a later study by Baker [Bak94] has revealed a major drawback of *DBT*. According to Baker, when no specific ordering heuristic is used, *DBT* performs better than *CBJ*. However, when ordering heuristics which are known to improve the run-time of *CSP* search algorithms are used [HE80, BR96, DF02], the performance of *DBT* is slower than the performance of *CBJ*. This phenomenon is easy to explain. Whenever the algorithm performs a back-jump it actually takes a variable which was placed according to the heuristic in a high position and moves it to a lower position. Thus, while in *CBJ*, the variables are ordered according to the specific heuristic, in *DBT* the order of variables becomes dependent upon the backjumps performed by the algorithm [Bak94].

In order to leave the assignments of non conflicting variables without a change on backjumps, *DBT* maintains a system of eliminating explanations (*Nogoods*) [Gin93]. As a result, the *DBT* algorithm maintains dynamic domains for all variables and can potentially benefit from the *Min-Domain* (fail first) heuristic. The present paper demonstrates empirically that this is the best performing version of *DBT*.

The *DBT* algorithm was combined with constraints propagation algorithms in order to increase its efficiency. The most successful version reported was *MAC-DBT*. The *MAC-DBT* algorithm uses *support lists* as in the well known *AC4* algorithm [MH86, BFR95], in order to maintain *Arc Consistency* throughout search. According to [JDB00] *MAC-DBT* outperforms versions of *DBT* which use a lower level of propagation methods (i.e. Forward Checking). Furthermore, *MAC-DBT* was also reported to outperform former versions of the *MAC* algorithm [JDB00, BR96].

The present study investigates the *DBT* algorithm from a number of aspects. First, we show that the advantage of *MAC-DBT* over *FC-DBT* holds only for a static ordering. When dynamic ordering heuristics are used, *FC-DBT* outperforms *MAC-DBT*. Second, we prove theoretically that a combined version of *DBT* that uses both *FC* and *MAC* performs equal or less computation than *MAC-DBT* as presented in [JDB00]. Our empirical results show an advantage of the combined version over *MAC-DBT*. Third, we present a version of *MAC-DBT* which does not preserve assignments which were jumped over (as in standard *CBJ*). This turns out to be the best performing version of *MAC-DBT*, which we term *MAC-CBJ-NG*. It benefits from the *Min-domain* heuristic, due to its maintenance of relevant *Nogoods*. Unlike standard *MAC-DBT* it does not harm dynamic ordering by keeping the jumped-over variables assigned. An analogous version of *FC-DBT* is *FC-CBJ-NG*. These versions preserve the properties of the ordering heuristic but in contrast to standard *CBJ* do not restore removed values whose *Nogoods* are consistent with the partial assignment. These two versions were found to run faster than all previous versions of *DBT*.

2 Constraint Satisfaction Problems

A *Constraint Satisfaction Problem (CSP)* is composed of a set of n variables V_1, V_2, \dots, V_n . Each variable can be assigned a single value from a discrete finite domain. Constraints or **relations** R are subsets of the Cartesian product of the domains of constrained variables. For a set of constrained variables $\{V_i, V_j, \dots, V_m\}$, with domains of values for each variable $\{D_i, D_j, \dots, D_m\}$, the constraint is defined as $R \subseteq D_i \times D_j \times \dots \times D_m$. A binary constraint R_{ij} between any two variables V_j and V_i is a subset of the Cartesian

product of their domains; $R_{ij} \subseteq D_j \times D_i$.

An assignment (or a label) is a pair $\langle var, val \rangle$, where var is a variable and val is a value from var 's domain that is assigned to it. A *partial solution* is a consistent set of assignments of values to a set of variables. A **solution** to a *CSP* is a partial solution that includes assignments to all variables [DF02].

3 MAC-DBT

Dynamic Backtracking (*DBT*) was first introduced in [Gin93]. *DBT* improves on *CBJ* by enabling variables which were jumped over on a backjump to keep their assignments. The *DBT* algorithm is described next, followed by a description of the later version of [JDB00] which combined *MAC* with *DBT*.

3.1 Dynamic Backtracking

The dynamic backtracking (*DBT*) algorithm is presented following [Bak94]. We assume in our presentation that the reader is familiar with *CBJ* [Pro93].

Like any backtrack algorithm, *DBT* attempts to extend a *partial solution*. A partial solution is an ordered set of value assignments to a subset of the *CSP* variables which is consistent (i.e. violates no constraints). The algorithm starts by initializing an empty partial solution and then attempts to extend this partial solution by adding assigned variables to it. When the partial solution includes assignments to all the variables of the *CSP*, the search is terminated successfully.

In every step of the algorithm the next variable to be assigned is selected according to the heuristic in use, and the values in its current domain are tested. If a value is in conflict with a previous assignment in the partial solution, it is removed from the current domain and is stored together with its eliminating *Nogood*. Otherwise, it is assigned to the variable and the new assignment is added to the partial solution [Bak94, Gin93].

An order is defined among the assignments in the partial solution. In the simplest form, this order is simply the order in which the assignments were performed (other options will be discussed).

Following [Gin93, Bak94], all *Nogoods* are of the following form:

$$(v_1 = q_1) \wedge \dots \wedge (v_{k-1} = q_{k-1}) \Rightarrow v_k \neq q_k$$

The left hand side serves as the explanation for the invalidity of the assignment on the right hand side. An eliminating *Nogood* is stored as long as its left hand side is consistent with the current partial solution. When a *Nogood* becomes invalid, it is discarded and the forbidden value on its right hand side is returned to the current domain of its variable [Gin93].

When a variable's current domain empties, the eliminating *Nogoods* of all its removed values are resolved and a new *Nogood* which contains the union of all assignments from all *Nogoods* is generated. The new *Nogood* is generated as follows. The right hand side of the generated *Nogood*, includes the assignment which was ordered last in the union of all *Nogoods* (the culprit assignment). The left hand side is a conjunction of the rest of the assignments in the united set [Gin93, Bak94].

After the new *Nogood* is generated, all *Nogoods* of the backtracking variable which include the culprit assignment are removed and the corresponding values are re-

turned to the current domain of the backtracking variable. Notice that this assignment to the backtracking variable cannot possibly be in conflict with any of the assignments which are ordered after the culprit assignment. Otherwise, they would have been included in an eliminating *Nogood*. The culprit variable on the right hand side of the generated *Nogood* is the next to be considered for an assignment attempt, right after its newly created *Nogood* is stored. Its new position in the order of the partial solution is after the latest assignment (i.e. it is moved to a lower place in the order than the position it had before).

The variables that were originally assigned after the culprit variable stay assigned. That is in contrast with *CBJ* (or *FC-CBJ* [Pro93]) that discards these assignments.

3.2 DBT with MAC

Enhancing a backtracking algorithm with look ahead methods (*FC*, *AC*) provides a significant improvement in performance [Pro93, KvB97, BR96, CvB01] and a solid basis for the ordering heuristics which use the size of domains of unassigned variables [HE80, BR96].

As mentioned above, the *DBT* algorithm was enhanced with *MAC* in [JDB00]. In *MAC-DBT*, after each assignment to a variable v_i , all the values left in the current domain of v_i are entered into a queue (Q) and then an arc consistency method based on *AC4* is performed. Explanations for the removal of values from the domains of unassigned variables are stored as in standard *DBT*. For a detailed description of *MAC-DBT* the reader is referred to the original paper [JDB00]. Following [JDB00], in our implementation for *MAC* we used the methodology of *AC4*. That means that we hold a support list for each $[variable, value]$ pair. While this approach is considered less attractive in general for *MAC* than the most cutting edge algorithms for arc consistency (*AC7* for example ensures the space complexity of *AC3* while preserving the time complexity of *AC4* [BFR95]). The use of support lists is essential in *MAC-DBT* for the generation of *Nogood* explanations for values which are removed as a result of *AC*. The explanation for a value removal is explained by the explanations for the removal of its supporters in the corresponding list.

4 Improving MAC-DBT

The contribution of the present paper is centered on improving the *MAC-DBT* algorithm in two different directions which are combined into a single algorithm. The first improvement addresses the weakness of the *DBT* algorithm reported by [Bak94] when ordering heuristics are used. It proposes a version of *DBT* that does not keep the assignments of “jumped-over” variables. The second improvement addresses the *MAC* method of [JDB00] and proposes a combined version of *FC* and *MAC* to improve its run time.

4.1 CBJ-NG

The *MAC-DBT* algorithm of [JDB00] still suffers from the phenomenon reported by [Bak94], that the *DBT* algorithm abolishes the benefits of the variable ordering heuristic used. Standard *CBJ* combined with look ahead methods, avoids this problem by discarding

the assignment of variables that it jumps-over. This way, the order of variables according to the desired heuristic is preserved. However, in contrast to *DBT*, in standard *CBJ* explanations for the removal of values are not stored and therefore on backtracks the entire domain of a variable which was jumped over is restored [Pro93].

The first improvement we propose is to combine *DBT* and *CBJ* into *CBJ-NG*. This algorithm benefits from the dynamic ordering of *CBJ* and from the maintenance of *Nogood* explanations of *DBT*. The algorithm uses one of two look ahead methods, *FC* or *MAC*, and the resulting algorithms are termed *FC-CBJ-NG* and *MAC-CBJ-NG*, respectively.

During the search process of *FC-CBJ-NG* and *MAC-CBJ-NG* variables are selected to be assigned according to a heuristic. Conflicting values of unassigned variables are filtered out by *forward checking* or by *AC*. These values are associated with *explicit* nogoods (*Nogoods* which derive explicitly from the problem's initial constraints) that explain their immediate removal as in *DBT* [Gin93].

When a current domain of an unassigned variable is exhausted, the *Nogood* explanations of its removed values are resolved and the result is a generation of an *inexplicit* (resolved) *Nogood*. Among the assignments of variables that appear in the generated *Nogood*, the assigned variable that is last ordered in the partial assignment is selected as the target of the backtrack (the *culprit*).

If the culprit is not the last assigned variable, the assignments of variables that were assigned after the culprit are discarded and their assigned values are returned to the current domains of these variables. Eliminating explanations that contain removed or discarded assignment are discarded as well, and the values that were eliminated by them are also returned to their current domains.

In contrast to standard *CBJ*, values of variables, whose assignment was discarded and their *Nogood* explanation is still consistent with the resulting partial assignment, are *not* returned to their variable's current domain. This forms a solid basis for ordering heuristics that are based on dynamic domain sizes of unassigned variables [HE80, BR96].

4.2 FC-CBJ-NG and MAC-FC-CBJ-NG

The pseudo code of *FC-CBJ-NG* and *MAC-CBJ-NG* is presented in Algorithms 1, 2 and 3. The *italic* lines are code that one needs to perform in order to transform the *FC-CBJ-NG* algorithm into *MAC-FC-CBJ-NG*. The pseudo code of both algorithms is described next.

Procedure **initialize** (line 2) performs initialization of the algorithms' data structures. Most of these operations are basic and technical, therefore they are not described in detail. If *MAC-FC* is performed, the **initialize** procedure computes the support lists for all values [MH86]. In this phase an empty domain generates a report that there is no solution [MH86].

The variables that are handled in lines 3-5 are assumed to be global and accessible by all procedures. *assigned*: is a stack that holds the already assigned variables in a LIFO order of their assignment. *unassigned*: is a pool of the unassigned variables. The variable *Pseudo* is initially pushed into the stack, in order to simplify the code. The *unassigned* pool contains all the variables that participate in the problem. *consistent* is a boolean variable which indicates whether the problem is consistent.

The loop in lines 6-16 follows the standard form of [Pro93] for *CSP* search algorithms. Arc consistency is applied before labeling (line 8). The procedure **label**(*var*) (Algorithm 2) tries to assign one value *val* from the *current_domain* of the unassigned variable *var*. If the assignment is successful, consistency remains true, and the assigned variable is entered into the *assigned* stack. Conflicting values are removed from the current domains of future variables. When *MAC* is used their respective pairs are inserted into the global queue *Q* for *AC* inspection. It is done by the procedure **check_forward**. If the assignment fails (a domain of an unassigned variable empties), state restoration is handled by procedure **undo_reductions**, and *val* is removed from the current domain of *var* along with its eliminating *Nogood*. When *MAC* is performed the pair (*var*, *val*) is inserted into the reduction set of the variable that is last assigned. The reduction set of a variable contains removed values whose eliminating *nogoods* are consistent with the assignment. These eliminating *Nogoods* are identified only after the assignment of that variable. The pair (*var*, *val*) is also inserted into *Q*, as the removal of *val* may cause the problem to violate the arc consistency property. At the end, the current domain of *var* is inspected for consistency (e.g non-emptiness).

Algorithm 1 MAC-FC/FC-CBJ-NG

```

1: procedure MAC-FC/FC-CBJ-NG
2:   initialize()
3:   unassigned  $\leftarrow$  variables
4:   assigned  $\leftarrow$  pseudoVar
5:   consistent  $\leftarrow$  true
6:   while unassigned.size() > 0 do
7:     if consistent then
8:       consistent  $\leftarrow$  check_AC()
9:     end if
10:    if consistent then
11:      next_var  $\leftarrow$  select_next_var(unassigned)
12:      consistent  $\leftarrow$  label(next_var)
13:    else
14:      consistent  $\leftarrow$  unlabel()
15:    end if
16:  end while
17:  report solution
18: end procedure

```

Procedure **unlabel** (in Algorithm 2) generally removes the assignment of an assigned variable. A value is removed if the current domain of an unassigned variable empties. The *Nogood* that explains this removal is resolved (line 2). If the *Nogood* is empty, the algorithm terminates unsuccessfully (a no solution is reported) (lines 3-6). Otherwise, the right hand side *RHS* variable, which is called *culprit*, is unassigned. The assignments of variables ordered after the culprit variable are discarded. In the **repeat** loop, these variables are extracted from the *assigned* stack and a restoration operation is performed by a call to **undo_reductions**. Note that when a variable is extracted its reduction set is unified with the reduction set of the former assigned variable. This ensures that all reduced values whose eliminating *Nogoods* remain consistent from the time that the culprit was assigned - when the problem was arc consistent

Algorithm 2 Procedures Label and Unlabel

```
1: procedure LABEL(var)
2:   select value from var.current_domain
3:   var.assignment  $\leftarrow$  value
4:   consistent  $\leftarrow$  check_forward(var)
5:   if not consistent then
6:     remove var.assignment from var.current_domain
7:     nogood  $\leftarrow$  resolve_nogoods(empty_domain_var)
8:     store(nogood)
9:     undo_reductions(var)
10:    lastAssigned  $\leftarrow$  assigned.head()
11:    add (var,var.assignment) to lastAssigned.reduction
12:    Q  $\leftarrow$  (var,var.assignment)
13:    if var.current_domain =  $\phi$  then
14:      empty_domain_var  $\leftarrow$  var
15:    end if
16:  else
17:    unassigned.remove(var)
18:    assigned.push(var)
19:  end if
20:  return var.current_domain  $\neq \phi$ 
21: end procedure

1: procedure UNLABEL
2:   nogood  $\leftarrow$  resolve_nogoods(empty_domain_var)
3:   if nogood =  $\phi$  then
4:     report no solution
5:     stop
6:   end if
7:   culprit  $\leftarrow$  nogood.RHS_variable
8:   remove var.assignment from var.current_domain
9:   store(nogood)
10:  repeat
11:    var  $\leftarrow$  assigned.pop()
12:    undo_reductions(var)
13:    unassigned.add(var)
14:    lastAssigned  $\leftarrow$  assigned.head()
15:    add var.reduction to lastAssigned.reduction
16:    if var  $\neq$  culprit then
17:      var.reduction  $\leftarrow \phi$ 
18:    else
19:      add (culprit,culprit.assignment) to lastAssigned.reduction
20:      add culprit.reduction to lastAssigned.reduction
21:      Q  $\leftarrow$  culprit.reduction
22:      culprit.reduction  $\leftarrow \phi$ 
23:      if culprit.current_domain =  $\phi$  then
24:        empty_domain_var  $\leftarrow$  culprit
25:      end if
26:      return culprit.current_domain  $\neq \phi$ 
27:    end if
28:  until var = culprit
29: end procedure
```

Algorithm 3 Procedures FC and check_AC

```
1: procedure CHECK_FORWARD(var1)
2:   foreach(var2 ∈ unassigned)
3:     foreach(val2 ∈ var2.current_domain)
4:       if not check(var1, var1.assignment, var2, val2) then
5:         remove val2 from var2.current_domain
6:         nogood ← (var1 = var1.assignment → var2 ≠ val2)
7:         store(nogood)
8:         Q.add((var2, val2))
9:         if var2.current_domain =  $\phi$  then
10:          empty_domain_var ← var2
11:          return false
12:        end if
13:      end if
14:    return true
15: end procedure

1: procedure UNDO_REDUCTIONS(culprit)
2:   foreach(var ∈ unassigned)
3:     foreach(val ∈ var.domain/var.current_domain)
4:       nogood ← store.get(var, val)
5:       if nogood.contains(culprit) then
6:         store.remove(nogood)
7:         remove((var, val) from culprit.reduction           ▷ if exists..
8:         insert val into var.current_domain
9:       end if
10:  end procedure

1: procedure CHECK_AC                                     ▷ used only in MAC
2:   while  $Q \neq \phi$  do
3:     (var1, val1) ← Q.extract()
4:     foreach(var2 ∈ set of variables constrained with var1)
5:       if var2 ∈ unassigned then
6:         foreach(val2 ∈ support(var1, var2, val1))
7:           if support(var2, var1, val2) ∩ var1.current_domain =  $\phi$  then
8:             remove val2 from var2.current_domain
9:             nogood ← resolve_nogood(support(var2, var1, val2))
10:            store(var2, nogood)
11:            lastAssigned ← assigned.head()
12:            add (var2, val2) to lastAssigned.reduction
13:            Q.add((var2, val2))
14:          end if
15:        if var2.current_domain =  $\phi$  then
16:          empty_domain_var ← var2
17:          return false
18:        end if
19:      end if
20:    end while
21:    return true
22: end procedure
```

- are accumulated. The respective pairs of these values are inserted into Q, and will be examined before the next assignment if the current domain of the culprit variable is consistent.

Procedure **undo_reductions** removes eliminating nogoods that contain the given unassigned variable in their explanations. It returns the previously eliminated values to the current domains of the variables they belong to. If MAC is performed these values are removed from the reduction set of the given variable.

Procedure **check_AC** is similar to the main procedure of AC4 [MH86]. It extracts pairs of the form (var, val) from the global queue Q until the queue is emptied. Values of the *current_domain* of unassigned variables, constrained with var , are checked for support (compatible value) in the *current_domain* of var . If there is no support for a value val' of a variable var' , then val' is removed and the *Nogood* is resolved. In this case the pair (var', val') is inserted into Q. The pair is also added to the reduction set of the last assigned variable. The procedure stops if it finds an empty *current_domain* or if the Q is emptied. This implies that the problem is arc consistent.

4.3 FC saves computation

Let us analyze the computational advantage of performing *FC* separately, as in Algorithm 1, over the version proposed in [JDB00]. The equivalence of the two methods generates the same removals in unassigned variables. Denote by t , the amount of computations spent by *AC4* when iterating over the values that are removed from the *current_domains* of unassigned variables, as a result of direct conflict with the assignment of v_i (all values that would have been removed by *FC*). Consider the set of all values that belong to *current_domains* of unassigned variables constrained with v_i . Now, consider a division of this set into two non intersecting subsets, S - the current set of values that are compatible with the assignment of v_i , and R - the current set of values that are in conflict with that assignment. The *FC* procedure visits each value in the *current_domains* of the unassigned variables just once, therefore the time spent by applying *FC* and than *AC* is:

$$t + |R| + |S|$$

The method for maintaining *AC* that is proposed in [JDB00] performs *FC* by inserting all unassigned values of the *current_domain* of variable i into the Q of removed values. We denote by d , the number of values that were inserted into Q . Each "removed" value triggers a check of support for each $val \in S$. Each $val \in R$ is examined with respect to the number of supporters that are retained in the *current_domain*(v_i). This number is greater or equal to 1. Assuming that the expected number of values checked (larger or equal to the number of supporters) for each $val \in R$, before it is removed from the current domain is k , we conclude that the computations performed are at least:

$$t + d \cdot |S| + k \cdot |R|$$

The cases in which $d = 0$ can be easily checked in advance (no computation needed during search) therefore the values of d and k are always greater or equal to 1. We can easily see that applying *FC* separately, as proposed in the present paper, generates less or equal computation than the method proposed in [JDB00].

5 Correctness of *MAC-FC-CBJ-NG*

Let us first assume the correctness of the standard *DBT* algorithm (as proved in [Gin93]) and prove that after the addition of forward checking, of MAC and the elimination of assignments after each backtrack, it is still sound, complete and it terminates.

Soundness is immediate since after each successful assignment the *partial_solution* is consistent. Therefore, when the *partial_solution* includes an assignment for each variable the search is terminated and a consistent solution is reported. \square

As in the case of standard *DBT* and *MAC-DBT*, the completeness of *MAC-FC-CBJ-NG* derives from exploring the entire search space except for sub search spaces which were found not to contain a solution. One needs to prove that the sub search spaces which *MAC-FC-CBJ-NG* does not search do not contain solutions. Sub search spaces are pruned by *Nogoods*. It is enough to prove the consistency of the set of *Nogoods* generated by *MAC-FC-CBJ-NG*. In other words, that the assignment of values removed by *Nogoods* never leads to solutions. For standard *DBT* this is proven in the original paper [Gin93].

The consistency property of *Nogoods* generated by *MAC-FC-CBJ-NG* can be shown as follows. First, observe that during the forward-checking and arc-consistency operations, *Nogoods* are standardly stored as explanations to removed values in the domain of *future variables*. Next, consider the case of a backtrack. It is easy to see that *Nogoods* of the *future variables* are resolved identically to those of standard *DBT*. Each *Nogood* is either an explicit *Nogood*, which is actually a constraint of the original problem, or a resolved *Nogood* which is a union of explicit *Nogoods*. In both cases any assignment which includes such a *Nogood* cannot be part of a solution. This proves the completeness of *MAC-FC-DBT*. \square

Last, we need to prove that the algorithm terminates. To this end we need to prove that the algorithm cannot enter an infinite loop. In other words, that a partial assignment cannot be produced by the algorithm more than once. We prove by induction on the number of variables of the *CSP*, n . For a *CSP* with a single variable, each of the values is considered exactly once. Assuming correctness of the argument for *CSPs* with k variables, for all $k < n$, we prove that in the case of a *CSP* with n variables the argument is still valid. Given a *CSP* of size n we assign the first variable and prune the inconsistent values of the unassigned variables using *FC* and *MAC*. The induced *CSP* is of size $n - 1$ in which according to the induction assumption the same assignment would not be generated twice. After the search of this induced *CSP* is completed, the result can be either a solution to the complete *CSP*, a non solution as a result of the production of an empty *Nogood* or a *Nogood* which includes the assignment of the first variable alone. In the first two cases we are done. In the third case, after the first variable replaced its assignment, it will never assign this value again. Therefore, none of the previous partial assignments can be produced again. This is true for each of the assignments of the first variable. \square

6 Experimental Evaluation

The common approach in evaluating the performance of *CSP* algorithms is to measure time in logical steps to eliminate implementation and technical parameters from affecting the results. We present results in both number of constraints checks and in CPU

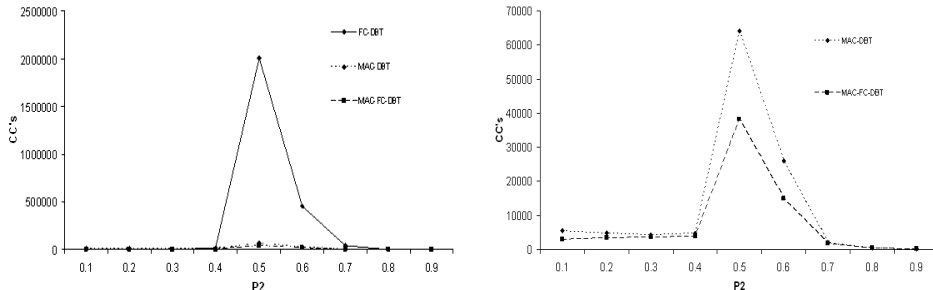


Figure 1: Constraints checks performed by *FC-DBT*, *MAC-DBT* and *MAC-FC-DBT* on low density CSPs ($p_1 = 0.3$) with static ordering.

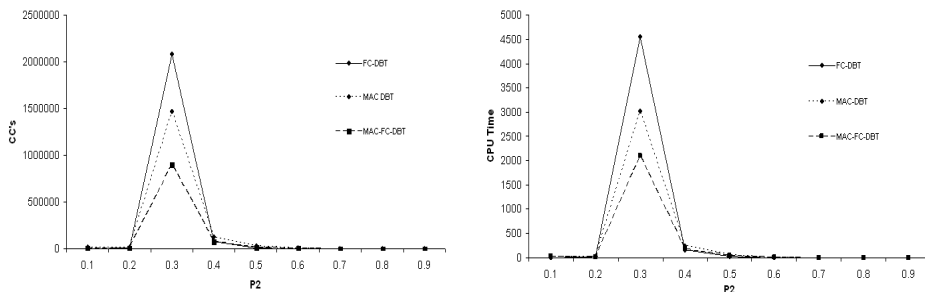


Figure 2: Same as Figure 1 for high density CSPs ($p_1 = 0.7$).

time [Pro96, KvB97].

The experiments were conducted on two problem scenarios: Random *CSPs* and on structured problems that represent a realistic scenario - Meeting Scheduling Problems [GW99].

Random *CSPs* are parametrized by n variables, k values in each domain, a constraints density of p_1 and a tightness p_2 which are commonly used in experimental evaluations of CSP algorithms [Smi96]. Two sets of experiments were performed on random problems. Both were conducted on *CSPs* with 20 variables ($n = 20$) and 10 values in the domain of each variable ($k = 10$). Two values of constraints density were used, $p_1 = 0.3$ and $p_1 = 0.7$. The tightness value p_2 , was varied between 0.1 and 0.9, in order to cover all ranges of problem difficulty. For each of the pairs of fixed density and tightness (p_1, p_2), 50 different random problems were solved by each algorithm and the results presented are an average of these 50 runs.

In the first set of experiments, *DBT* with three different lookahead methods was compared, Forward Check (*FC-DBT*), MAC (*MAC-DBT*) and a combined lookahead version that use both FC and MAC (*MAC-FC-DBT*).

The left hand side (LHS) of Figure 1 presents the number of constraints checks performed by the three versions of the algorithm with static ordering on low density *CSPs* ($p_1 = 0.3$). *MAC-DBT* outperforms *FC-DBT*, as reported by [JDB00]. The algorithm proposed in the present paper (that performs *FC* separately of *AC4*) *MAC-FC-DBT* outperforms both *MAC-DBT* and *FC-DBT*. A closer look at the difference

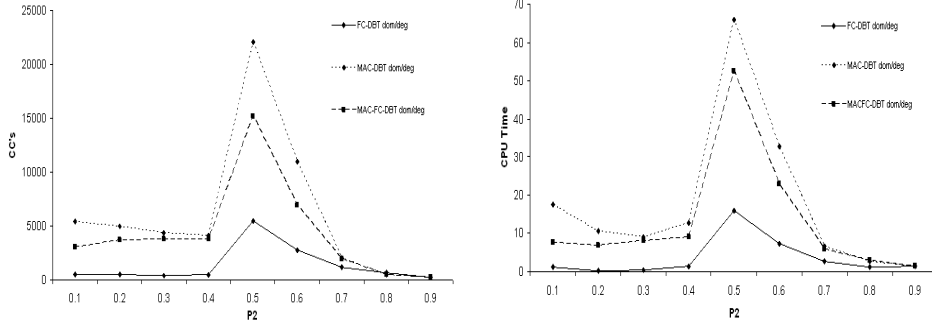


Figure 3: Constraints checks and CPU-time performed by *FC-DBT*, *MAC-DBT* and *MAC-FC-DBT* with *dynamic* (min-domain / degree) ordering ($p_1 = 0.3$).

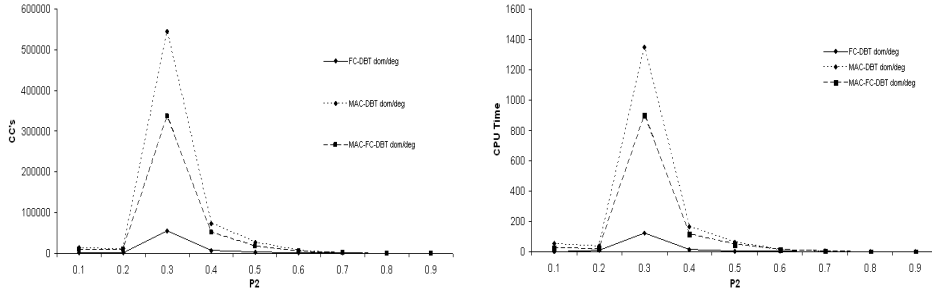


Figure 4: Same as Figure 3 for high density CSPs ($p_1 = 0.7$).

between *MAC-FC-DBT* and *MAC-DBT* is presented on the right hand side (RHS) of the figure. Note that in this experiment all algorithms perform *DBT* and not *CBJ-NG*. They all preserve the jumped over assignments on a backtrack as in [Gin93, Bak94, JDB00].

The LHS of Figure 2 presents similar results for high density CSPs ($p_1 = 0.7$). In this case the difference between *FC-DBT* and *MAC-DBT* is much smaller. The RHS of Figure 2 presents the same results in *CPU* time and is presented in order to show the similarity between these two measures.

Figure 3 presents a comparison between the same versions of the algorithm when using the min-domain/degree heuristic [BR96]. The results in constraints checks (LHS) and in *CPU* time (RHS) show clearly the advantage of *FC-DBT* on both versions of *MAC-DBT* when ordering heuristics are used. Similar results for high density CSPs are presented in Figure 4. The difference in favor of *FC* is higher on dense CSPs.

In the second set of experiments *FC-DBT* and *MAC-FC-DBT* are compared with the well known *FC-CBJ* algorithm and with the two proposed versions which perform backjumping as in *CBJ* but store *Nogoods* as in *DBT* (see Section 3.1), our proposed algorithms *MAC-FC-CBJ-NG* and *FC-CBJ-NG* (Algorithms 1, 2 and 3). All algorithms use the min-domain/degree heuristic.

Figure 5 presents a comparison between different versions of the algorithm on low density random *CSP*s. The results show a large difference between the *FC* algorithms and the *MAC* algorithms. However, there are small differences between the different

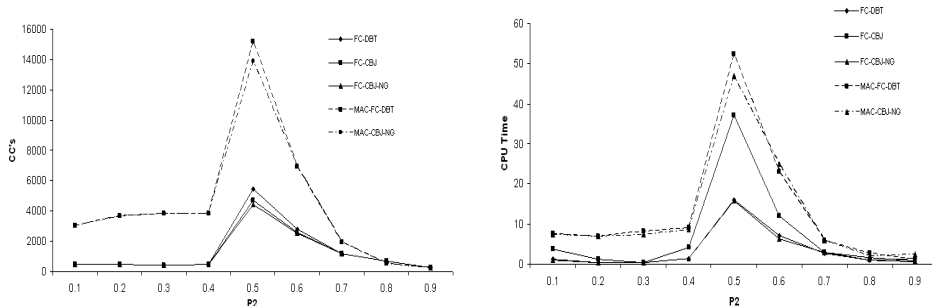


Figure 5: Constraints checks and CPU-time for low density CSPs ($p_1 = 0.3$).

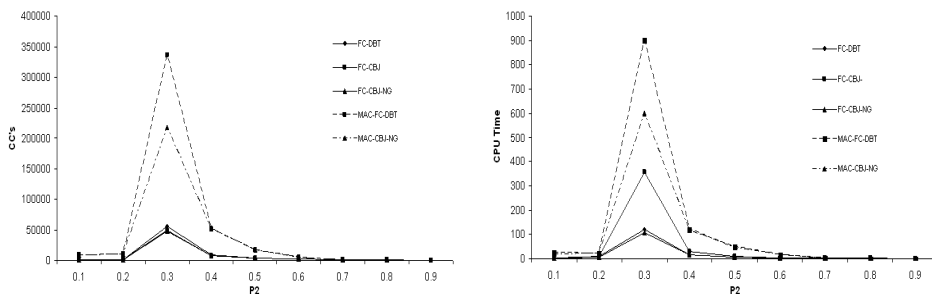


Figure 6: Same as Figure 5 for high density CSPs ($p_1 = 0.7$).

versions. Similar results were obtained for high density random *CSPs* (Figure 6).

In the third set of experiments, the algorithms are compared when solving structured problems, Meeting Scheduling [GW99]. In this special class of problems, variables represent meetings between agents. Arrival constraints exist between meetings of the same agent. The tightness of the problem grows with the number of meetings per agent [GW99].

Figure 7 presents the performance of the algorithms, solving random meeting scheduling problems with 40 meetings (variables), domain size of 12 time slots, 18 agents and arrival constraints which were randomly selected between 2 to 6 [GW99]. The results in constraints checks (LHS) and in CPU-time are again very similar. On structured problems the differences between the different versions of *MAC-DBT* are much smaller than the differences between the different versions of *FC*. Still, the best performing algorithm is the *FC* version which performs *CBJ* and uses *DBT Nogoods*. To emphasize its advantage, the results of the most successful versions of *MAC* and *FC* are presented in Figure 8. In the case of structured problems the largest difference in performance is for tight problems.

7 Discussion

Dynamic Backtracking (*DBT*) was proposed by [Gin93] as a mechanism that enables the search algorithm to perform backjumping while preserving the assignments of variables which were jumped over. The proposed algorithm was found to *harm* the effect

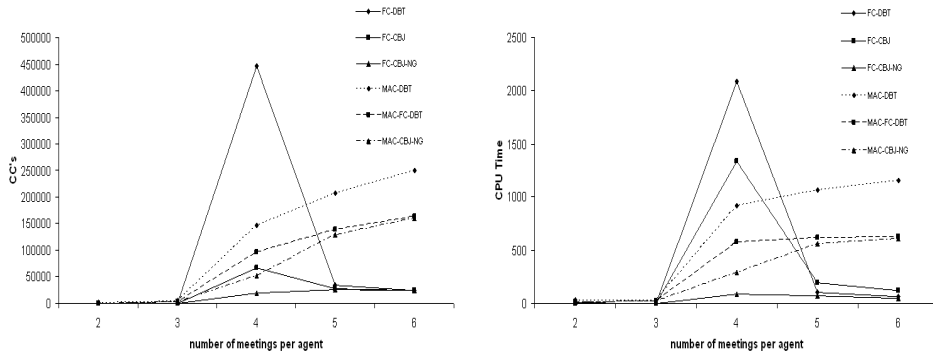


Figure 7: Constraints checks and CPU-time performed by the different algorithms on Random Meeting Scheduling Problems.

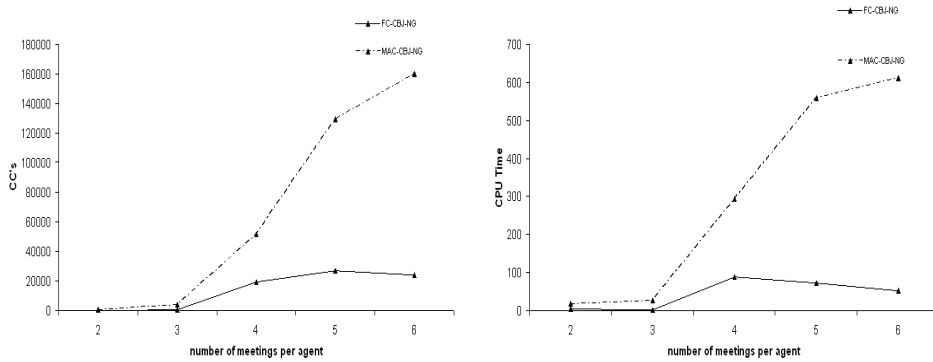


Figure 8: Constraints checks and CPU-time performed by the best versions of *MAC* and *FC* on Random Meeting Scheduling Problems.

of ordering heuristics and to perform poorly compared to standard conflict directed backjumping when powerful ordering heuristics are used [Bak94].

Enhancing *DBT* with local consistency methods (*FC*, *MAC*) was proposed by [JDB00] who found that the most successful version is *MAC-DBT*. This version was also found to outperform standard versions of *MAC*[MOSHE PLEASE ADD THE REFERENCE].

The results presented in the present paper demonstrate that the advantage of *MAC-DBT* over *FC-DBT* exists *only when static order is maintained*. When dynamic ordering heuristic is used, *FC-DBT* runs faster than *MAC-DBT*. Our presented results, both theoretical and empirical, show clearly the advantage of performing a combined version of *MAC-DBT* with explicit forward-checking, over the *MAC-DBT* of [JDB00].

The best performing algorithms presented in this paper do not preserve assignments which were jumped over (as in [Gin93]). As a result, the properties of the ordering heuristic are preserved in contrast to standard *DBT*. Both versions of *DBT* with lookahead (*FC* and *MAC*) benefit from using the *DBT* mechanism for storing *Nogood* explanations. The benefit arises from the ability to determine which value should be restored to a variable's domain. Updated domains during backjumping enhance ordering heuristics that are based on domain size. The algorithms proposed by the present paper *FC-CBJ-NG* and *MAC-CBJ-NG*, improve on both *MAC-DBT* and *FC-*

DBT. The best performing algorithm was found to be *FC-CBJ-NG*. Its advantage over all other versions is most pronounced on structured (*Meeting Scheduling*) problems.

References

- [Bak94] Andrew B. Baker. The hazards of fancy backtracking. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI '94), Volume 1*, pages 288–293, Seattle, WA, USA, July 31 - August 4 1994. AAAI Press.
- [BFR95] C. Bessière, E. Freuder, and J. Régin. Using inference to reduce arc consistency computation. In *IJCAI-95*, pages 592–598, 1995.
- [BR96] C. Bessiere and J.C. Regin. Mac and combined heuristics: two reasons to forsake fc (and cbj?) on hard problems. In *Proc. CP 96*, pages 61–75, Cambridge MA, 1996.
- [CvB01] X. Chen and P. van Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research (JAIR)*, 14:53–81, 2001.
- [Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufman, 2003.
- [DF02] R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136:2:147–188, April 2002.
- [Gin93] M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.
- [GW99] I.P. Gent and T. Walsh. Csplib: a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from <http://csplib.cs.strath.ac.uk/>. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).
- [HE80] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
- [JDB00] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.
- [KvB97] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 21:365–387, 1997.
- [MH86] Roger Mohr and Thomas C. Henderson. Arc and path consistence revisited. *Artif. Intell.*, 28(2):225–233, 1986.
- [Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
- [Pro96] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.
- [Smi96] B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155 – 181, 1996.